Arrays, variables, axes and missing values

What is geospatial data?

Thinking about data…

Axes

Time, Level, Lat, Lon, other….

Data arrays

[0,0,0,1,1,1,0,0,,101, 01,01,1010] x lots

Metadata specific to Axes and Data

Global Info (metadata)

Source, institute, dataset id, history of conversions…

Some philosophy behind the design

- The basic unit of data in CDMS is the ***variable – a multidimensional array,*** augmented with a ***domain*** and with ***metadata***.
- The ***domain*** describes the spatial location and temporal information associated with the array.
- The ***metadata*** associated with a variable consists of a collection of *attribute-value* pairs (such as ***units="m***").
- A variable may be stored in a single physical ***file*** (of varying formats) or in a collection of files, called a ***dataset***.

Some philosophy behind the design

- In CDAT, variables can be used much like arrays. i.e. you can use them directly in calculations as you would in IDL, Matlab etc.
- By working with CDMS variables for computation the associated domain and metadata information is carried along with the computation.
- Benefit in data readability/portability – easily interpreted by software/people later on. E.g. plotters can realise map projections.

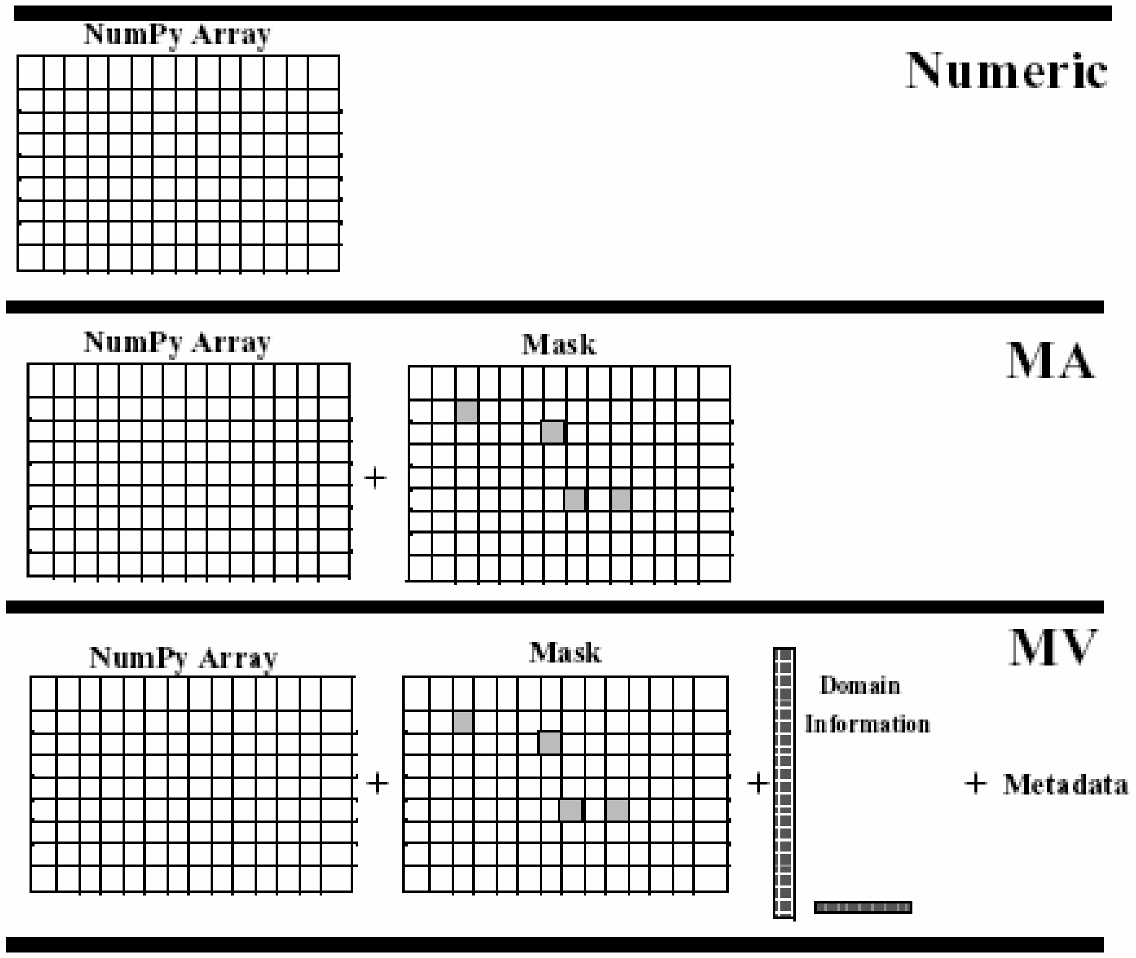
The hierarchy of arrays (into variables)

- To be truly extensible and interoperable, CDAT provides a hierarchy of representations for data arrays (allowing interfacing with non-CDAT packages):

1. **Numeric Array:** a multidimensional array, all elements have the same data type (real, integer etc.). From *Numeric Python* (*numpy*).
2. **Masked Array (MA):** A Numeric array with an optional missing data mask. Operations on these compute the mask of the result.
3. **Masked Variable (MV):** Masked array with domain and metadata. A masked variable in memory is referred to as ***transient variable*** and a masked variable in a dataset is called a ***file variable.***

Some philosophy behind the design

FIGURE 1. Relationships between array abstractions.



The Numeric module

- This brings us to a fundamental underlying component of CDAT: ***The Numeric module***
- This Package brings real arithmetic into Python, Numeric array are similar to C (or Fortran) arrays, they also re-introduce the single precision **int** and **float.**
- Documentation can be found at: http://www.pfdubois.com/numpy/html2/numpy.html
- **NOTE**: On your pythonic travels you may come across **numarray**, a next generation of **Numeric**, with bells and whistles!

Working with Numeric arrays

- Creating a Numeric array is easy

```
import Numeric
a=Numeric.array([1,2,3,4,5,6,7,8,9,10,11,12])
b=Numeric.array([[1.,2,3],[4,5,6],[7,8,9],[10,
  11,12]], ‘f’) # ‘f’ designates float type.
```

- To determine the shape/rank (number of dimensions) of an array:

```
print a.shape # (12,)
print b.shape # (4,3)
print Numeric.rank(b) # 2
```

Numeric array types

- To determine the “type” of an array:

```
print a.typecode() # returns ‘l’ i.e “long”
print b.typecode() # ‘d’ i.e. “double”
```

To convert an array type:

```
c=a.astype(‘d’)
```

Available typecodes: (Numeric.

Complex, Complex0, Complex8, Complex16, Complex32, Complex64, Float, Float0, Float8, Float16, Float32, Float64, Int, Int0, Int8, Int16, Int32, UnsignedInteger, UnsignedInt8, UnsignedInt16, UnsignedInt32)

- To determine if an object is a Numeric array:

```
isinstance(b,Numeric.ArrayType)
type(b)==Numeric.ArrayType
```

Array operations

- Operations can be applied directly and propagate through all dimension (NO NEED for LOOPS):

```
>>> import Numeric
>>> a=Numeric.array([2,3,4,5], "f")
>>> b=Numeric.array([9,8,7,6], "f")
>>> c=a+b
>>> c
[ 11., 11., 11., 11.,]
```

- Most function are applied to the first axis (0) by default but can be applied to another axis by passing an extra argument:

```
c=Numeric.average(b) # [5.5,6.5,7.5]
c=Numeric.average(b,1) # [2.,5.,8.,11.] # Average
   after axis “1”
```

Numeric - Some useful functions

Functions include:

- **average**(array)
- **sum**(array) # sums the contents of an array
- **where**(c1, array1, array2) # or (c1, int, int)

  e.g. To convert all values of 50 to -9999 in an array:

  ```
  where(greater(arr,50), -9999, arr)
  ```
- greater, less, greater_equal, less_equal, equal, not_equal
- logical_and, logical_or, logical_not
- absolute, sqrt, power, exp, log, log10
- sin, cos, tan, arcsin, arctan, etc…
- **arrayrange**(first, last, stride, typecode)
- **maximum**(a, b)**, minimum**(a, b) # returns max or min of a and b

  ```
  >>> print maximum(array([2,3,100]), 50)
  [ 50, 50,100,]
  ```

Numeric - More useful functions

- **sort**(x) – sorts contents of array
- **ravel**(x) – returns array flattened to 1-dimension (like “x.flat”)
- **rank**(x) – gets rank of array
- **resize** (x, new_shape) - returns a new array with specified shape.
- **reshape** (x, new_shape) - returns a copy of x with the given new shape.
- **transpose** (a, axes=None) - performs a reordering of the axes depending on the tuple of indices axes; the default is to reverse the order of the axes.
- **compress**(condition, x, dimension = -1) - those elements of x corresponding to those elements of condition that are "true". condition must be the same size as the given dimension of x.
- **concatenate** (arrays, axis=0) concatenates the arrays along the specified axis.
- **argsort**()/**searchsorted**()
- **take**()

Numeric arrays functions

- The following function are tied to Numeric arrays objects:
  - `a.byteswapped()` # switch between little and big endian
  - `a.copy()`
  - `a.iscontiguous()` # are all elements contiguous in memory ?
  - `a.itemsize()` # Return the size in bytes of a single element of the array
  - `a.resize()` # Do NOT use, use the resize function from Numeric
  - `a.savespace(1/0)` # Will preserve the type of the array no matter what or turn that off
  - `a.spacesaver()` # is the above function on ?
  - `a.tolist()` # converts array to list type
  - `a.toscalar()` # Return first data point
  - `a.typecode()` # Return typecode of the array

Undefined ‘Missing’ Values: Masked Array (MA)

- Unfortunately in our field, most of the time the dataset we’re dealing with is “incomplete” either because it contains observations or simply because no data exist at a specific point (interpolation below ground, ocean model not computing over land, …)

- In order to overcome this deficiency, the Numeric module is supplemented with a similar Package: **MA** which stands for **Masked Array**.

- MA has most of numeric functionalities but also knows how to handle missing values.

MA-specific functions (1)

`MA._get_print_limits()/MA._set_print_limits()` # prints the number of values included 300 default

`MA.getmask(array)` # Return mask associated with array (None if no mask)

`MA.masked_greater(array,array2 or value),` … # Mask array where array is greater/less… than array2 or value passed

`MA.masked_inside/masked_outside(array,a1/v1,a2/v2)` # creates an array with values inside/outside the closed interval [v1, v2] masked. v1 and v2 may be in either order.

MA-specific functions (2)

`MA.masked_where(condition, data, copy=1)`

Creates a masked array whose shape is that of condition, whose values are those of data, and which is masked where elements of condition are true. Condition can be something like “`MA.greater(data, value)`”.

```
MA.masked_equal/masked_values(data, value,
   rtol=1.0000000000000001e-05, atol=1e-08, copy=1,
   savespace=0)
```

Creates a masked array where value=value; mask is None if possible. If copy==0, and otherwise possible, result may share data values with original array.

Let d = filled(data, value). Returns d masked where:

abs(data-value)<=atol + rtol * abs(value)

`MA.mask_or(a1,a2)`

Create a mask using a1 values or a2 values (if a1 value is None). Use None if they are both None.

Adding metadata: Axes and Attributes

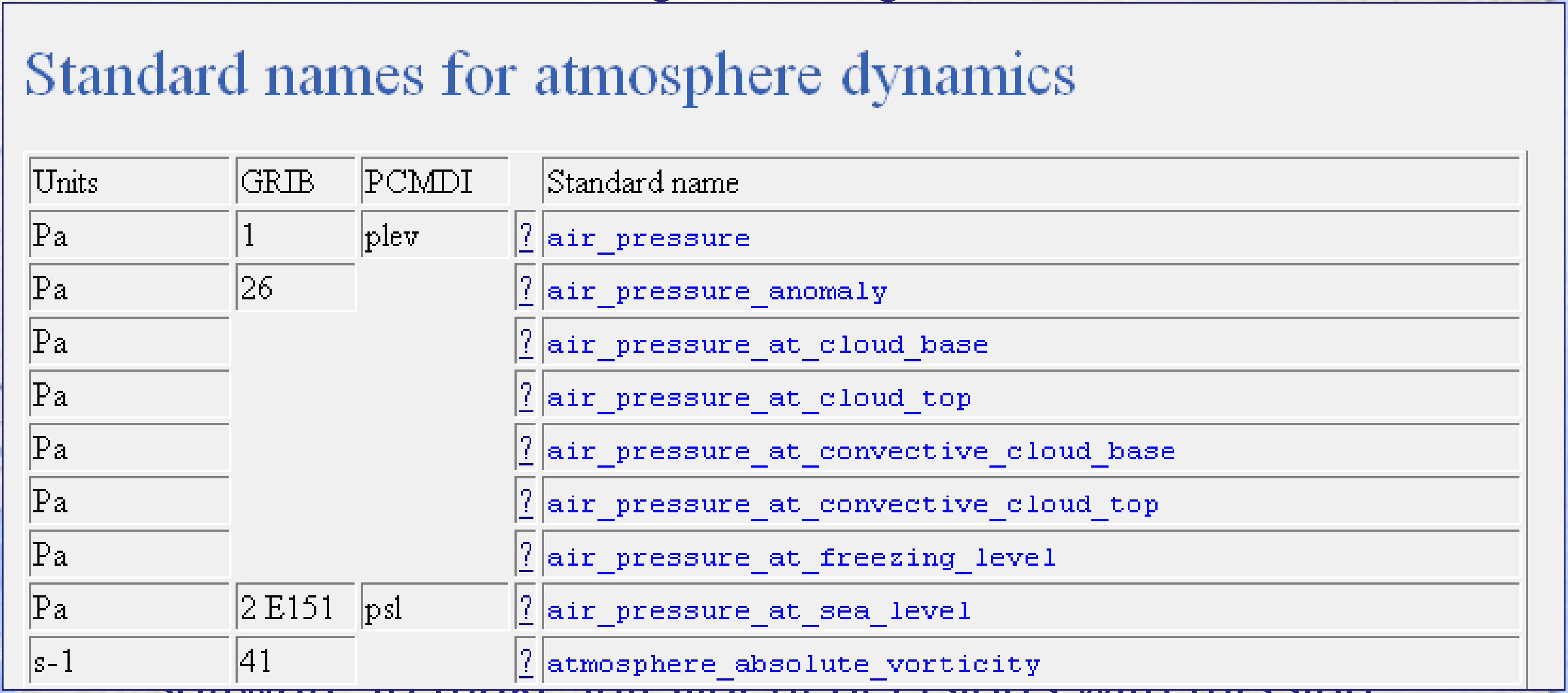
- Being able to deal with missing values isn’t necessarily enough. In our field most of the time we need to know “more” about the data, the domain spanned (spatially and in time), some specific attribute (name, history, etc…)

- All these are called “**metadata**”.

- CDAT manages metadata according to the **Climate and Forecasts (CF) Metadata Convention** for NetCDF. This provides a set of rules for producing *well-formed* data files.

Why the CF Metadata Convention?

- Standardisation is a good thing!



Standard names for atmosphere dynamics

Units	GRIB	PCMDI		Standard name
Pa	1	plev	?	air_pressure
Pa	26		?	air_pressure_anomaly
Pa			?	air_pressure_at_cloud_base
Pa			?	air_pressure_at_cloud_top
Pa			?	air_pressure_at_convective_cloud_base
Pa			?	air_pressure_at_convective_cloud_top
Pa			?	air_pressure_at_freezing_level
Pa	2 E151	psl	?	air_pressure_at_sea_level
s-1	41		?	atmosphere_absolute_vorticity

software to make intelligent decisions with missing values, plotting, diagnostics and sub-setting.

Advice on *well-formed* CF-compliant NetCDF

- For guidance on writing *well-formed* NetCDF (which CDAT will mainly do for you) see Unidata's web page:
  http://my.unidata.ucar.edu/content/software/netcdf/BestPractices.html
- It provides guidance on:
  - Coordinate systems
  - Variable groupings
  - Variable attributes
  - Calendar Date/Time
  - and more...
- But the CF document should be your guiding light:
  http://www.cgd.ucar.edu/cms/eaton/cf-metadata/index.html

Seal of approval?

The IPCC is requiring all data providers to upload CF-compliant data.

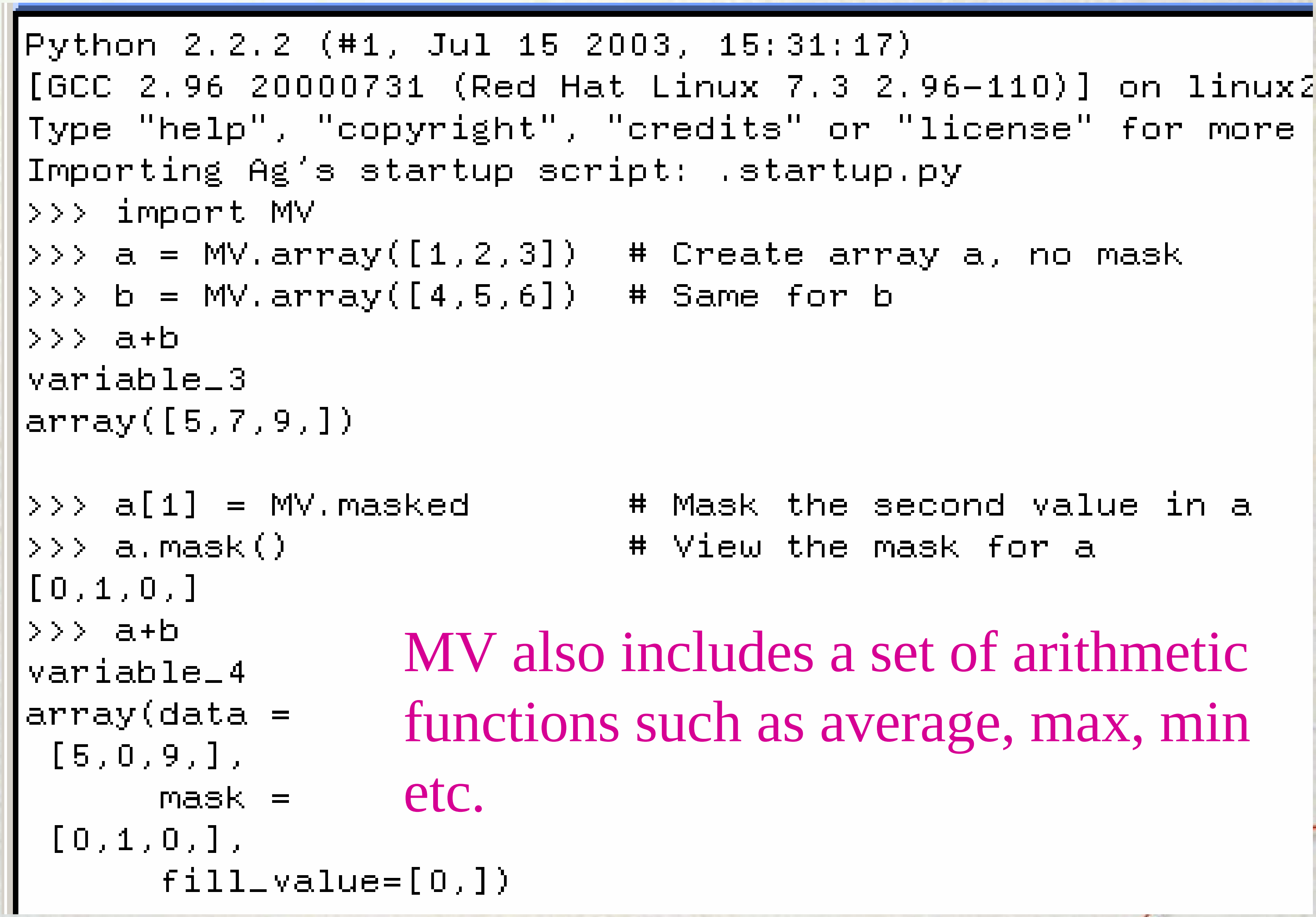
Adding metadata: Enter the Masked Variable (MV)

- Numeric and MA do not know about metadata, therefore came the need for a new module (**Masked Variable (MV)**) which can retain such information.
- MV is defined in the **cdms** package so masked variables are available when you import cdms.
- Masked Variables have additional functionalities tied to the **cdms** package, such as getAxis, getLatitude, getGrid, regrid, subdomain extraction, etc…

3 Types of variables

- A variable can be obtained either from a file, a collection of files (a dataset), or as the result of computation. Correspondingly there are three types of variables in CDAT:
  - A ***file variable*** is a variable associated with a **single data file**. Interaction involves I/O operations.
  - A ***dataset variable*** is a variable associated with a collection of files or dataset (normally described by one CDML file). Dataset variables are read-only.
  - A ***transient variable*** is an ‘in-memory’ object not associated with a file or dataset. You can compute transient variables from other variables or build them yourself.

The MV Module in action



```
Python 2.2.2 (#1, Jul 15 2003, 15:31:17)
[GCC 2.96 20000731 (Red Hat Linux 7.3 2.96-110)] on linux2
Type "help", "copyright", "credits" or "license" for more
Importing Ag's startup script: .startup.py
>>> import MV
>>> a = MV.array([1,2,3])   # Create array a, no mask
>>> b = MV.array([4,5,6])   # Same for b
>>> a+b
variable_3
array([5,7,9,])

>>> a[1] = MV.masked        # Mask the second value in a
>>> a.mask()                # View the mask for a
[0,1,0,]
>>> a+b
variable_4
array(data =
 [5,0,9,],
      mask =
 [0,1,0,],
      fill_value=[0,])
```

MV also includes a set of arithmetic functions such as average, max, min etc.

Interrogating a CDAT file/dataset

- You have already seen how to open a CDAT file and extract variables. Here are some other useful things you can get from a file/dataset object:

  Remember: you can list the methods using “dir(<object>)”.

  `f=cdms.open(‘afile’)` # Assign “f” as cdms file object

  `f.id` # returns the file/dataset name

  `f.grids` # returns the grids in a file

  `f.variables` # returns the variables in a file

  `f.axes` # returns the axes in a file

  `f.attributes` # returns all the attributes (including axes) in a file

  `f.getdimensionunits('longitude')` # returns the units of the dimension

  `f.getVariable(‘temp‘)` # returns a variable from a file

  `f.listglobal()` # returns a list of global file attributes

  `f.getglobal(‘source’)` # returns the value of the specified attribute

Selecting variables from a file with ( ) or [ ]

- The commonest way to get to variables in a file is to call the file object using normal brackets:

  ```
  f=cdms.open(‘afile’)
  var=f(‘temperature’)
  ```

- When dealing with enormous files (or datasets – made up of potentially 100s of 1000s of files) you don’t want to extract all the data (memory won’t let you), so you need a way of finding out about a variable to decide what you need.

- Use square brackets to extract the metadata* only (i.e. the attributes, grids, axes etc):

  ```
  var_metadata=f[‘temperature’]
  ```

* This is actually a pointer to the file variable.

Re-ordering variable axes on input

- Here is a useful piece of functionality - re-ordering the axes when you read a variable from a file:
  - standard "tzyx" order (as held in *most* files):

```
>>> f('r')
r  array(
 array (1,21,73,144) , type = f, has 220752 elements)
```

  - "xytz" ordering:

```
>>> f('r', order="xy")
r  array(
 array (144,73,1,21) , type = f, has 220752 elements)
```

  - "ytxz" ordering:

```
>>> f('r', order="ytxz")
r  array(
 array (73,1,144,21) , type = f, has 220752 elements)
```

Interrogating the variable metadata (1)

- From your variable object you might want to find out:
  - What axes is this variable defined against?

    ```
    >>> var.getAxisList() # to see all of them
    >>> var.getLongitude() # longitude only
    >>> var.getLongitude()[:] # longitude values
    # var.getTime(), var.getLevel() – similar
    >>> var.getGrid() # grid (if appropriate)
    ```

  - What shape is the variable?

    ```
    >>> var.shape
    ```

  - What is the size (number of values) and rank of this variable?

    ```
    >>> var.size() ; var.rank()
    ```

Interrogating the variable metadata (2)

- What is the missing value?

  ```
  >>> var.getMissing()
  ```

- What attributes exist for this variable?

  ```
  >>> var.listattributes()
  ```

- What is the value of attribute ‘name’?

  ```
  >>> var.getattribute(‘name’)
  ```

- What is the axis order of this variable?

  ```
  >>> var.getOrder()
  ```

- What is all the metadata for this variable?

  ```
  >>> var.attributes
  ```

Interrogating axes and grids (1)

- From your axis object you might want to find out:
  - What does this axis look like?

```
>>> ax=var.getAxis(2)
>>> ax
   id: latitude
   Designated a latitude axis.
   units:  degrees_north
   Length: 73
   First:  -90.0
   Last:   90.0
   Other axis attributes:
      axis: Y
   Python id:  40ba476c
```

Interrogating axes and grids (2)

– What are the units?

```
>>> ax.units
```

– What are the actual values?

```
>>> ax.getValue() # or ax[:]
```

– Is it time? Is it latitude?

```
>>> ax.isTime() ; ax.isLatitude()
```

– What are the bounds (if they exist)?

```
>>> ax.getBounds()
```

– What is the key metadata for this axis?

```
>>> ax.listall()
```

– Is it a circular axis (i.e. longitude wraps around itself)?

```
>>> ax.isCircular()
```

Special methods on time axes

- Time axes have a number of specific methods:
  - Show axis as component time list?

```
>>> ax.asComponentTime()
```

  - Show axis as relative time list?

```
>>> ax.RelativeTime()
```

  - What is the calendar?

```
>>> ax.getCalendar()
```

Sub-setting and *squeezing* the actual data

- As we’ve already seen, when you want to subset data you can just specify the spatial and temporal region you want (and you can keep doing it…):

```
>>> import cdms
>>> f=cdms.open(‘file1.nc’)
>>> var=f(‘temp’, time=(“1999-1”, “1999-2”))
>>> slab1=var(level=16, latitude=(0, 90))
>>> slab2=slab1(latitude=(30,40))
>>> slab3=slab2(longitude=2)
# Note that you still have a 4-D variable,
# You might want to remove the singular axes:
>>> slab4=slab3(squeeze=1)
# squeeze also comes in handy when plotting
```

Using “slice” to subset

- Python has an object called a ***slice*** that is made up of three integers:
  - start
  - end
  - step

- Invoke a slice with:
  - slice([start,] stop[, step])

```
s=slice(0, 100, 5) # 0 to 100 step 5
s2=slice(50)         # 0 to 50 step 1
```

Using “slice” to subset

- Why are slices useful in CDAT? Use them to sub-select data:
  - When grabbing data from a file or dataset:

    ```
    x=f('slhf', time=slice(0, 1200, 12),
                                  lat=slice(0,180,5))
    ```

  - When grabbing data from a variable:

    ```
    x_subset=f('slhf', lon=slice(0,360,10))
    ```

  - Both of these return a ***transient*** (in memory) **variable**.

Selectors – another way of sub-setting

- The **Selector** class lives in the cdms.selectors module. It allows you to pre-define a selector that can then be re-used in code:

```
from cdms.selectors import Selector
sel1 = Selector(time=('1979-1-1','1979-2-1'),
  level=1000.)
x1 = v1(sel1)
x2 = v2(sel1)
```

- Pre-defined selector slices for axes:

```
from cdms import timeslice, levelslice
x = hus(timeslice(0,2), levelslice(16,17))
```

- Or you can use the **domain** selectors in **cdutil**:

```
from cdutil.region import *
NH=NorthHemisphere=domain(latitude=(0.,90.))
SH=SouthHemisphere=domain(latitude=(-90.,0.))
```

Writing as opposed to reading?

- We have just seen a number of methods for examining existing metadata. To create your own metadata and data you'll need:

```
cdms.createAxis()
cdms.createVariable()
```

- Then many of the methods we've seen have opposites:

```
var.setMissing(miss_value) # instead of 'get'
var.setattribute(name, value) # "    "    "    "
ax.setBounds(newBounds)       # "    "    "    "
ax.setCalendar(newCal)        # "    "    "    "
ax.designateLatitude() # instead of 'is'
```

- Use **"`dir(var)`"** and **"`dir(ax)`"** to list the methods and then **"`help(var.methodname)`"** to see how it should be used.

Creating a good axis from scratch

- Create an array from a list or Numeric:

```
values=range(0,360,5)
lon=cdms.createAxis(values)
```

- You could stop here, but we like metadata! So designate it:

```
lon.designateLongitude()
```

- And name, units…

```
lon.id="longitude"
lon.standard_name="longitude"
lon.units="degrees_east"
lon.comment="This really is longitude!"
```

Creating a CDMS variable (1)

- You need to use **`cdms.createVariable():`**

```
cdms.createVariable(array,
  typecode=None, copy=0, savespace=0,
  mask=None, fill_value=None,
  grid=None, axes=None,
  attributes=None, id=None)
```

- See the CDMS manual for a full explanation of the options:

http://esg.llnl.gov/cdat/cdms.pdf

Creating a CDMS variable (2)

- Having created your axes:

```
# Let’s say “arr” is an array converted from your
# fortran programme.
>>> print arr.shape # to check
(12, 60, 181, 360) # time/lev/lat/lon
# You have defined some axes as well...
>>> print len(ti), len(depth), len(lat), len(lon)
12 60 181 360  # Good, they match!
```

- Let’s set up an attribute dictionary in advance:

```
>>> atts=={"long_name":"Atlantic meridional
    overturning streamfunction", "name":"Atlantic
    meridional overturning streamfunction",
    "units":"sverdrups"}
```

- Then create the transient variable:

```
>>> amos=cdms.createVariable(arr, id=“amos”,
          axes=[ti, depth, lat, lon],
          attributes=atts, fill_value=1.e-20)
```

Missing values and CF

And a word about missing values:

- The CF convention says that we should encode missing values with the "**_FillValue**" attribute.
- In python the leading underscore makes the attribute "private" and so it doesn't get automatically written to a file with the variable.
- CDAT therefore uses the now deprecated (in CF) **"missing_value"**:

```
>>> var.setMissing(1.e-20)
>>> print var.missing_value
1.e-20
```

- Note that you can also use **valid_min, valid_max** and **valid_range** for missing ranges.